MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms

Daniel Gracia Pérez
LRI, Paris Sud/11 University
INRIA Futurs, France
HiPEAC network
gracia@lri.fr

Gilles Mouchard
CEA LIST
LRI, Paris Sud/11 University
INRIA Futurs, France
HiPEAC network
mouchard@lri.fr

Olivier Temam
LRI, Paris Sud/11 University
INRIA Futurs, France
HiPEAC network
temam@lri.fr

Abstract

While most research papers on computer architectures include some performance measurements, these performance numbers tend to be distrusted. Up to the point that, after so many research articles on data cache architectures, for instance, few researchers have a clear view of what are the best data cache mechanisms. To illustrate the usefulness of a fair quantitative comparison, we have picked a target architecture component for which lots of optimizations have been proposed (data caches), and we have implemented most of the performance-oriented hardware data cache optimizations published in top conferences in the past 4 years. Beyond the comparison of data cache ideas, our goals are twofold: (1) to clearly and quantitatively evaluate the effect of methodology shortcomings, such as model precision, benchmark selection, trace selection. . . , on assessing and comparing research ideas, and to outline how strong is the methodology effect in many cases, (2) to outline that the lack of interoperable simulators and not disclosing simulators at publication time make it difficult if not impossible to fairly assess the benefit of research ideas. This study is part of a broader effort, called MicroLib, an open library of modular simulators aimed at promoting the disclosure and sharing of simulator models.

1 Introduction

Simulators are used in most processor architecture research works, and, while most research papers include some performance measurements (often IPC and more specific metrics), these numbers tend to be distrusted because the simulator associated with the newly proposed mechanism is rarely publicly available, or at least not in a standard and reusable form; and as a result, it is not easy to check for design and implementation hypotheses, potential simplifications or errors. However, since the goal of most processor architecture research works is to *improve* performance, i.e., do better than previous research works, it is rather frustrating not to be able to clearly quantify the benefit of a new architecture mechanism with respect to previously proposed mechanisms. Many researchers wonder, at some point, how their mechanism fares with respect to previously proposed ones and what is the best mechanism, at least for a given processor architecture and benchmark suite (or even a single benchmark); but many consider, with reason, that it is excessively time-consuming to implement a significant array of past mechanisms based on the research articles only.

The purpose of this article is threefold: (1) to argue that, provided a few groups start populating a common library of modular simulator components, a broad and systematic quantitative comparison of architecture ideas may not be that unrealistic, at least for certain research topics and ideas; we introduce a library of modular simulator components aiming at that goal, (2) to illustrate this quantitative comparison using data cache research (and at the same time, we start populating the library on this topic), (3) to investigate the following set of methodology issues (in the context of data cache research) that researchers often wonder about but do not have the tools or resources to address:

- Which hardware mechanism is the best with respect to performance, power or cost?
- Are we making significant progress over the years?
- What is the effect of benchmark selection on ranking?
- What is the effect of the architecture model precision, especially the memory model in this case, on ranking?
- When programming a mechanism based on the article, does it often happen that we have to second-guess the authors' choices and what is the effect on mechanism performance and ranking?

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

- What is the effect of trace selection on ranking?

Comparing an idea with previously published ones means addressing two major issues: (1) how do we implement them? (2) how do we validate the implementations?

(1) The biggest obstacle to comparison is the necessity to implement again all the previously proposed and relevant mechanisms. Even if it usually means fewer than five mechanisms, we all know that implementing even a single mechanism can mean a few weeks of simulator development and debugging. And that is assuming we have all the necessary information for implementing it. Reverse-engineering all the implementation details of a mechanism from a 10-page research article can be challenging. An extended abstract is not really meant (or at least not usually written so as) to enable the reader to implement the hardware mechanism, it is meant to pass the idea, give the rationale and motivation, and convince the reader that it *can* be implemented; so some details are omitted because of paper space constraints or for fear they would bore the reader.

(2) Assuming we have implemented the idea presented in an article, then how do we validate the implementation, i.e., how do we know we have properly implemented it? First, we must be able to reconstruct exactly the same experimental framework as in the original articles. Thanks to widely used simulators like SimpleScalar [3], this has become easier, but only partially so. Many mechanisms require multiple minor control and data path modifications of the processor which are not always properly documented in the articles. Then, we need to have the same benchmarks (and benchmark traces), which is again facilitated by the Spec benchmarks [24], but they must be compiled with exactly the same compiler (e.g., the same gcc version) on the same platform. Third, we need to parameterize the base processor identically, and few of us specify all the SimpleScalar parameters in an article. Fortunately (from a reverse-engineering point of view) or unfortunately (from an architecture research point of view), many of us use many of the same default SimpleScalar parameters. Fourth, to validate an implementation, we need to compare the simulation results against the article numbers, which often means approximately reading numbers on a bar graph... And finally, since the first runs usually don't match, we have to do a combination of performance debugging and reverse-engineering of the mechanisms, based on second-guessing the authors' choices. By adding a dose of common sense, one can usually pull it off, but even then, there always remains some doubt, on apart of the reader of such a comparison, as to how accurately the researcher has implemented other mechanisms.

In this article, we illustrate these different points through data cache research. We have collected the research articles on performance improvement of data caches from the past four editions of the main conferences (ISCA, MICRO, ASPLOS, HPCA). We have implemented most of the mechanisms corresponding to pure hardware optimizations (we have not tried to reverse-engineer software optimizations). We have also implemented older but widely referenced mechanisms (*Victim Cache, Tagged Prefetching* and *Stride Prefetching*). We have collected a total of 15 articles, and we have implemented only 10 mechanisms either because of some redundancies among articles (e.g., one article presenting an improved version of a previous one), implementation or scope issues. Examples of implementation issues are the *data compression prefetcher* technique [28] which uses data *values* (and not only addresses) which are not available in the base SimpleScalar version, *eager write-back* [15] which is designed for and tested on memory-bandwidth bound programs which were not available; an example of scope issue is the *non-vital loads* technique [19] which requires modifications of the register file, while we decided to focus our implementation and validation efforts on data caches only.

It is possible that our own implementation of these different mechanisms is flawed because we have used the same error-prone process described in previous paragraphs; so the results given in this article, especially the conclusion as to which are the best mechanisms, should be considered with caution. On the other hand, all our models are available on the MicroLib library web site [1], as well as the ranking, so any researcher can check our implementation, and in case of inaccuracies or errors, we will be able to update the online ranking and the disseminated model.

Naturally, comparing several hardware mechanisms means more than just ranking them using various metrics. But the current situation is the opposite: researchers do analyze and compare ideas qualitatively, but they have no simple means for performing the quantitative comparisons.

This study is part of a broader effort called *MicroLib* which aims at facilitating the comparison and exchange of simulator models among processor architecture researchers. In Section 2 we describe our experimental framework, in Section 3, we attempt to answer the methodology questions listed above, and in Section 4 we present the *MicroLib* project.

2 Experimental Framework and Validation

2.1 Experimental framework

We used SimpleScalar for the comparison because it was also used in a large share of the mechanisms, though not in *Tagged Prefetching* [23], *Victim Cache* [13], *Stride Prefetching* [7], *Frequent Value Cache* [29], *Markov Prefetching* [12] and *Content-Directed Data Prefetching* [4]. We have stripped SimpleScalar of its cache and memory models, and replaced them with our MicroLib data

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

Parameter	Value
Processor core	
Processor Frequency	2 GHz
Instruction Windows	128-RUU, 128-LSQ
Fetch, Decode, Issue width	8 instructions per cycle
Functional units	8 IntALU, 3 IntMult/Div, 6 FPALU, 2 FPMult/Div, 4 Load/Store Units
Commit width	up to 8 instructions per cycle
Memory Hierarchy	
L1 Data Cache	32 KB/direct-mapped
L1 Data Write Policy	Writeback
L1 Data Allocation Policy	Allocate on Write
L1 Data Line Size	32 Bytes
L1 Data Ports	4
L1 Data MSHRs	8
L1 Data Reads per MSHR	4
L1 Data Latency	1 cycle
L1 Instruction Cache	32 KB/4-way associative/LRU
L1 Instruction Latency	1 cycle
L2 Unified Cache	1 MB/4-way associative/LRU
L2 Cache Write Policy	Writeback
L2 Cache Allocation Policy	Allocate on Write
L2 Line Size	64 Bytes
L2 Ports	1
L2 MSHRs	8
L2 Reads per MSHR	4
L2 Latency	12 cycles
L1/L2 Bus	32-byte wide, 2 Ghz
Bus	
Bus Frequency	400 MHz
Bus Width	64 bytes (512 bits)
SDRAM	
Capacity	2 GB
Banks	4
Rows	8192
Columns	1024
RAS To RAS Delay	20 cpu cycles
RAS Active Time	80 cpu cycles
RAS to CAS Delay	30 cpu cycles
CAS Latency	30 cpu cycles
RAS Precharge Time	30 cpu cycles
RAS Cycle Time	110 cpu cycles
Refresh	Avoided
Controler Queue	32 Entries

Table 1. Baseline configuration.

cache models. In addition to the various data cache models, we have developed and used an SDRAM model for most experiments. Note that a detailed memory model for SimpleScalar has been recently proposed [5] but it was not yet publicly distributed when we performed all experiments.

We have used *sim-outorder* of the SimpleScalar 3.0d suite [3] and the parameters in Table 1, which we found in many of the target articles [14, 10, 9]; they correspond to a scaled up superscalar implementation; the other parameters are set to their default SimpleScalar values. Though many of these articles use a constant 70-cycle SimpleScalar memory latency, we have opted for a slightly more realistic memory model, and implemented an SDRAM model with the timings indicated in Table 1. Section 3.3 presents the effect of the memory model (SimpleScalar memory model versus our SDRAM model) on the mechanisms performance.

We have compared the mechanisms using the SPEC CPU2000 benchmark suite [24]. The benchmarks were compiled for the Alpha instruction set using `cc` DEC C V5.9-008 on Digital UNIX V4.0 (Rev. 1229), `cxx` Compaq C++ V6.2-024 for Digital UNIX V4.0F (Rev. 1229), `f90` Compaq Fortran V5.3-915 and `f77` Compaq Fortran V5.3-915 compilers with SPEC peak settings. For each program, we used a 500-million instruction trace, skipping up to the first SimPoint [21]; the SimPoint has been extracted using the Basic Block Vector generator; 100-million SimPoint traces were already shown to give simulation results within 15% (for floating point benchmarks) to 18% (for integer benchmarks) of the full benchmark simulation [21].

2.2 Validating the Implementation

Validating a hybrid SimpleScalar+MicroLib model. Our cache architecture is different, and we believe slightly more realistic, than the SimpleScalar model; the differences are detailed below. We found an average 6.8% IPC difference between the hybrid SimpleScalar+MicroLib implementation and the original SimpleScalar implementation. We then progressively modified the SimpleScalar cache model to get closer to our MicroLib model in order to find the reasons for these performance differences, and in the same time, to ascertain that our model had no hidden performance bug. We found that most of the performance variation is due to the following architecture differences:

- The SimpleScalar MSHR (miss address file) has unlimited capacity; in our cache model its capacity parameters are defined in Table 1.
- In SimpleScalar, the cache pipeline is insufficiently detailed. As a result, a cache request can never delay next requests, while in a pipelined implementation, such delays can occur. Several events can delay a request: two misses on the same cache line but for different addresses can stall the cache, upon receiving a request the MSHR is not available for one cycle…
- The processor Load/Store Queue (LSQ) can always send requests to the cache in SimpleScalar, while the abovementioned cache stalls (plus MSHR full) can temporarily stall the LSQ.
- In SimpleScalar the refill requests (incoming memory request) seem to use additional cache ports. For instance, when the cache has two ports, it is possible to have two fetch requests and a refill request at the same time. We strictly enforce the number of ports, and upon a refill request, only one normal cache request can occur with two ports.

After altering the SimpleScalar model so that it behaves like our MicroLib model, we found that the average IPC difference between the two models was down to 2%, see Figure 1. Note that, we *do not* use the original SimpleScalar model, we use our MicroLib model.

Besides this comparison with SimpleScalar, we have performed additional validations by plugging the different

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

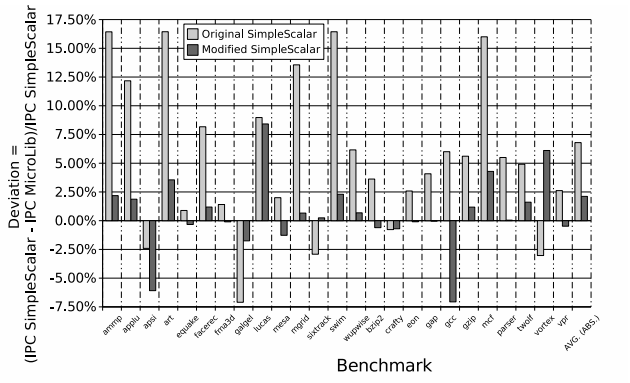

Figure 1. MicroLib cache model validation.

cache mechanisms within our own MicroLib superscalar processor model, called OoOSysC [1], which has the advantage of actually performing all computations. As a result, the cache not only contains the addresses but the *actual values* of the data, i.e., it really executes the program, unlike SimpleScalar. Confronting the emulator with the simulator for every memory request is a simple but powerful debugging tool.[1] For instance, in one of the implemented models, we forgot to properly set the dirty bit in some cases; as a result, the corresponding line was not systematically written back to memory, and at the next request at that address, the values differed.

Validating the implementation of data cache mechanisms. The most time-consuming part of this research work was naturally reverse-engineering the different hardware mechanisms from the research articles. The different mechanisms, a short description and the corresponding reference are listed in Table 2, and the mechanism-specific parameters are listed in Table 3. The Spec benchmarks used in each article are indicated in Table 4 (some articles naturally use non-Spec benchmarks).

For several mechanisms, there was no easy way to do an IPC validation. The metric used in *FVC* and *Markov* is miss ratio, so only a miss ratio-based validation was possible. *VC*, *TP* and *SP* have been proposed several years ago, so the benchmarks and the processor model differed significantly. *CDP* and *CDPSP* used a proprietary Intel simulator and their own benchmarks. For all the above mechanisms, the validation consisted in ensuring that absolute performance values were in the same range, and that tendencies were often similar (relative performance difference of architecture parameters, among benchmarks, etc. . .).

For *TK*, *TKVC* and *TCP* we used the performance graphs provided in the articles for the validation. For the validation process only, instead of the abovementioned SimPoint traces, we used 2-billion instruction traces, skipping the first billion, as in the articles. For the sake of the validation, and in this section only, we also use the original SimpleScalar 70-cycle constant latency memory model. Figure 2 shows the relative speedup error between the graph numbers and our simulations (note that some articles do not provide IPCs, but only speedups with respect to the base SimpleScalar cache configuration). The average error is 5%, but the difference can be very significant for certain benchmarks, especially $ammp$. In general, tendencies are preserved, but not always, i.e., a speedup or a slowdown in an article can become a slowdown or a speedup in our experiments, as, respectively, for *gcc* and *gzip* (for *TK*). Note that, three articles [14, 9, 10] use exactly the same SimpleScalar parameters of Table 1, even though the first mechanism was published in 2000 and the last one in 2003. Only the SimpleScalar parameters of *GHB* (not included in the graph of Figure 2), proposed at HPCA 2004, are different (140-cycle memory latency). In the next paragraph, we illustrate reverse-engineering issues with *DBCP*.

Parameter	Value
Victim Cache	
Size/Associativity	512 Bytes / Fully assoc.
Frequent Value Cache	
Number of lines	1024 lines
Number of frequent values	7 + unknown value
Timekeeping Cache	
Size/Associativity	512 Bytes/Fully assoc.
TK refresh	512 cpu cycles
TK threshold	1023 cycles
Markov Prefetcher	
Prediction Table Size	1 MB
Predictions per entry	4 predictions
Request Queue Size	16 entries
Prefetch Buffer Size	128 lines (1 KB)
Tagged Prefetching	
Request Queue Size	16
Stride Prefetching	
PC entries	512
Request Queue Size	1
Content-Directed Data Prefetching	
Prefetch Depth Threshold	3
Request Queue Size	128
CDP + SP	
SP PC entries	512
CDP Prefetch Depth Threshold	3
Request Queue Size (SP/CDP)	1/128
Timekeeping Prefetcher	
Address Correlation	8KB, 8-way assoc.
Request Queue Size	128 entries
Tag Correlating Prefetching	
THT size	1024 sets, direct-mapped, stores 2 previous tags
PHT size	8KB, 256 set, 8 way assoc.
Request Queue Size	128 entries
Dead-Block Correlating Prefetcher	
DBCP history	1K entries
DBCP size	2M 8-way
Request Queue Size	128 entries
Global History Buffer	
IT entries	256
GHB entries	256
Request Queue Size	4

Table 3. Configuration of cache optimizations.

[1] Besides debugging purposes, this feature is also particularly useful for testing value prediction mechanisms.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

Acronym	Mechanism	Description
TP	Tagged Prefetching [23] (L2)	One of the very first prefetching techniques: prefetches next cache line on a miss, or on a hit on a prefetched line.
VC	Victim Cache [13] (L1)	A small fully associative cache for storing evicted lines; limits the effect of conflict misses without (or in addition to) using associativity.
SP	Stride Prefetching [17] (L2)	Originally proposed by Chen and Baer [7]: an extension of tagged prefetching that detects the access stride of load instructions and prefetches accordingly.
Markov	Markov Prefetcher [12] (L1)	Records the most probable sequence of addresses and uses that information for target address prediction.
FVC	Frequent Value Cache [29] (L1)	A small additional cache that behaves like a victim cache, except that it is just used for storing frequently used values in a compressed form (as indexes to a frequent values table). The technique has also been applied, in other studies [28, 27], to prefetching and energy reduction.
DBCP	Dead-Block Correlating Prefetcher [14] (L1)	Records access patterns finishing with a miss and prefetches whenever the pattern occurs again.
TK	Timekeeping [9] (L1)	Determines when a cache line will no longer be used, records replacement sequences, and uses both information for a timely prefetch of the replacement line.
TKVC	Timekeeping Victim Cache [9] (L1)	Determines if a (victim) cache line will again be used, and if so, decides to store it in the victim cache.
CDP	Content-Directed Data Prefetching [4] (L2)	A prefetch mechanism for pointer-based data structures that attempts to determine if a fetched line contains addresses, and if so, prefetches them immediately.
CDPSP	CDP + SP (L2)	A combination of CDP and SP as proposed in [4].
TCP	Tag Correlating Prefetching [10] (L2)	Records miss patterns per tag and prefetches according to the most likely miss pattern.
GHB	Global History Buffer [17] (L2)	Records strides patterns in a load address stream and prefetches if patterns recur.

Table 2. Target data cache optimizations.

Mechanism	ammp	applu	apsi	art	equake	facerec	fma3d	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
DBCP	√			√	√														√		√					
TK/TKVC/TCP/DBCPTK	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
GHB	√			√				√	√		√		√	√	√			√			√	√				√

Table 4. Benchmarks used in validated mechanisms.

Further insights from the authors. Because one of the key points of our study is to argue that research articles may not provide sufficient information on experiments and methodology, we decided, on purpose, not to contact the authors in a first step, in order to assess how much we could dig from the research articles only.[2] Later on, we have either contacted the authors or have been contacted by authors, and tried to fix or further validate the implementation of their mechanisms. We have been in contact with the authors of *GHB*, *DBCP*, *TCP* and *CDPSP*. We had difficulties validating *DBCP* by ourselves in the first step, but later on, the authors of *DBCP* devoted special efforts to helping us fix our implementation. Together, we found that the issues were either related to ours misinterpreting the experimental setup or the description of the mechanism, or to missing information in the article. We briefly list them below as an illustration of reverse-engineering issues:

- Our initial implementation of *DBCP* was off by 30% from the original *DBCP* article. The *DBCP* article was using `pisa` benchmarks while we were using `alpha` benchmarks. We had not appropriately considered a footnote in the article saying "benchmarks were compiled for SimpleScalar". `alpha` benchmarks tend to generate more *DBCP* signatures (sequences of ld/st instruction addresses accessing a given cache line) than the `pisa` benchmarks, and since we use a correlation table of the same size (as in the *DBCP* article), `alpha` benchmarks performed worse.
- The number of entries in the correlation table we used was wrong (half the correct value) due to an incorrect interpretation of the text; as a result, many potentially useful predictions were lost.
- The original article omitted to mention that the correlation mechanism had to prehash the ld/st instruction addresses before xoring them with the signatures. This error produced aliasing conflicts in the correlation table, which, in turn, degraded the efficiency of prefetching.
- The article also omitted to mention that the confidence counters of signatures in the correlation table are decreased if the signature no longer induces misses. As a result, the correlation table was polluted with useless

[2] The submitted version of this article did not include any feedback from the authors.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

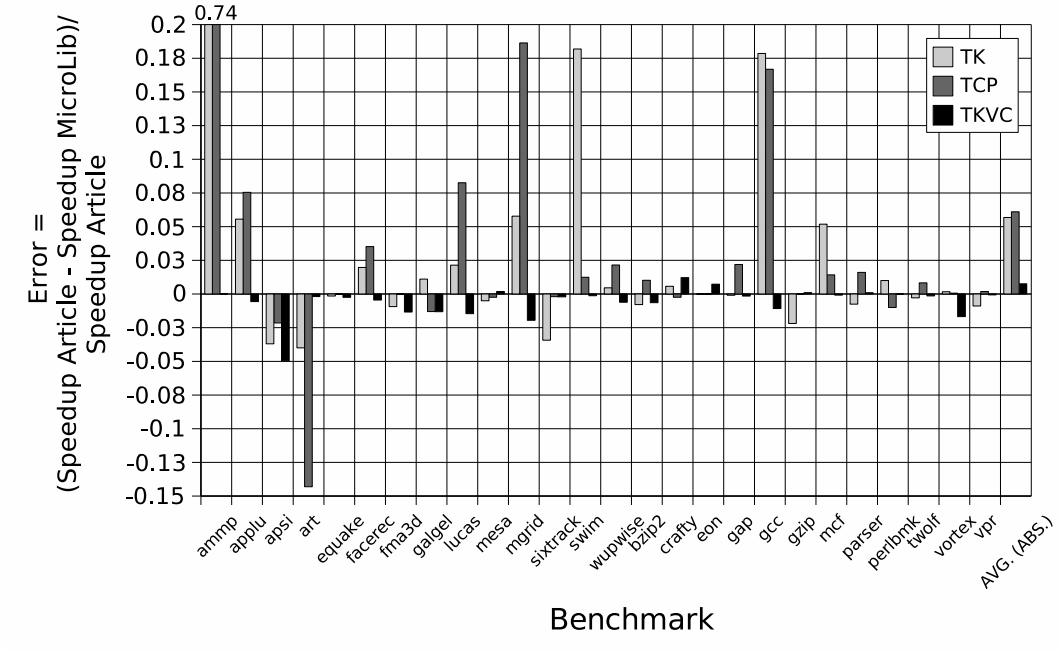

Figure 2. Validation of TK, TCP and TKVC.

signatures, again degrading the efficiency of prefetching.

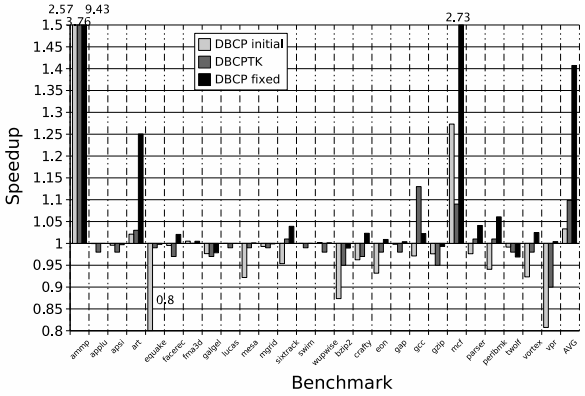

Figure 3. Fixing the DBCP reverse-engineered implementation.

Figure 3 shows the speedups obtained before (`initial`) and after (`fixed`) fixing our *DBCP* implementation, i.e., an average difference of 38%. The figure also contains the *DBCPTK* bar: these values have been extracted from the article which proposed *TK* [9] and which compared *TK* against *DBCP*; in other words, the authors of *TK* have reverse-engineered *DBCP* to perform a quantitative comparison.[3] Interestingly, their own reverse-engineering effort brought results close to our initial implementation, possibly because they may have made some of the same reverse-engineering errors. Moreover, in the *TK* article, *TK* outperforms *DBCP* by 6%, while our fixed implementation of *DBCP* outperforms our implementation of *TK* by 32%. Note that relative performance changes again with a different experimental setup (see the next section); note also that we have used the fixed *DBCP* implementation in the remainder of this article.

3 A Quantitative Comparison of Hardware Data Cache Optimizations

The different subsections correspond to the methodology questions listed in Section 1. Except for Section 3.1, all the comparisons relate to the IPC metric and are usually averaged over all the benchmarks listed in Section 2.1, except for Section 3.2.

3.1 Which hardware mechanism is the best with respect to performance, power and/or cost? Are we making any progress?

Performance. Figure 4 shows the average IPC speedup over the 26 benchmarks for the different mechanisms with

[3]Note that all the numbers in this graph were obtained using the same experimental setup as in the *TK* and *DBCP* articles for the sake of the validation: 2-billion traces skipping 1-billion, and 70-cycle SimpleScalar memory model. In the rest of the article, we used the experimental setup mentioned at the beginning of this section.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

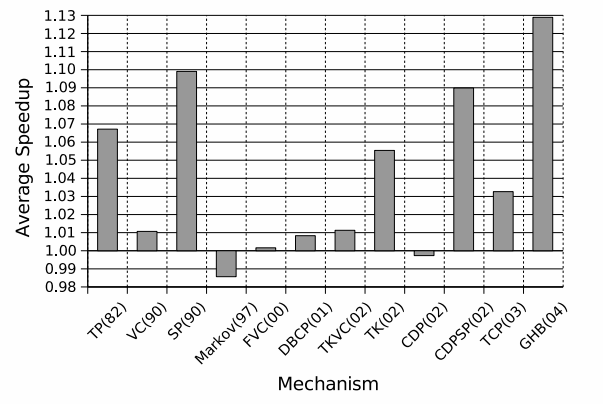

Figure 4. Speedup.

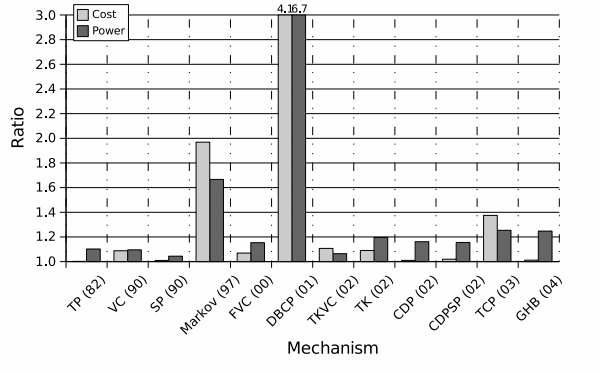

Figure 5. Power and Cost Ratios.

respect to the base cache parameters defined in Section 2.1. We find that the best mechanism is *GHB*, a recent evolution (HPCA 2004) of *SP*, an idea originally published in 1982, and which is the second best performing mechanism, then followed by *TK*, proposed in 2002. A very simple (and old) hardware mechanism like *TP* performs also quite well. Overall, it is striking to observe how irregularly performance has evolved from 1990 to 2004, when all mechanisms are considered within the same processor.

In more details, *FVC*, which was evaluated using miss ratios in the original article, seems to perform less favorably in a full processor environment. Though the overall performance of *CDP* (a prefetch mechanism for pointer-based data structures) seems rather poor, it does benefit to some pointer-based benchmarks like *twolf* (1.07 speedup) and *equake* (1.11 speedup). On the other hand, *CDP* also does degrade the performance of pointer-intensive benchmarks like *mcf* (0.75 speedup); in *ammp*, prefetch requests fetch 64-byte line, but the next pointer address is located 88-bytes down in the main data structure (`struct`), and thus *CDP* systematically fails to prefetch it, saturating the memory bandwidth with useless prefetch requests.

Finally, note also that the relative speedup differences for some of the mechanisms in Figure 4 is fairly close to the reverse-engineering error shown in Figure 2, suggesting that having systematic access to the original simulators (as opposed to building our own reverse-engineered versions) is important for a fair assessment of research ideas.

Cost. We evaluated the relative cost (chips area) of each mechanisms using CACTI 3.2 [22], and Figure 5 provides the area ratio (relative cost of mechanism with respect to base cache). Not suprisingly, *Markov* and *DBCP* have very high cost due to large tables, while other lightweight mechanisms like *TP*, or even *SP* and *GHB* (small tables) incur almost no additional cost. What is more interesting is the correlation between performance and cost: *GHB* and *SP* remain clear winners, closely followed by *TK*; when factoring cost, *TP* appears like a more attractive solution.

Power. We evaluated power using XCACTI [11]; Figure 5 shows the relative power increase of each mechanism. Naturally, power is determined by cache area and activity, and not surprisingly, *Markov* and *DBCP* have strong power requirements. In theory, a costly mechanism can compensate the additional cache power consumption with more efficient, and thus reduced, cache activity, though we found no clear example along that line.[4] Conversely a cheap mechanism with significant activity overhead can be power greedy. It is apparently the case for *GHB*: even though the additional table is small, each miss can induce up to 4 requests, and a table is scanned repeatedly, hence the high power consumption. In *SP*, on the other hand, each miss request induces a single request, and thus *SP* is very efficient, just like *TP*.

Best overall tradeoff (performance, cost, power). When power and cost are factored in, *SP* seems like a clear winner, *TK* and *TP* performing also very well. *TP* is the oldest mechanism, *SP* has been proposed in 1990 and *TK* has been very recently proposed in 2002. While which mechanism is the best very much depends on industrial constraints, it is still probably fair to say that the progress of data cache research over the past 20 years has been all but regular.

In the remaining sections, ranking is focused on performance (due to paper space constraints), but naturally, it would be necessary to come up with similar conclusions for power, cost, or all three parameters combined.

DBCP vs. Markov	TKVC vs. VC
TK vs. DBCP	CDP/CDPSP vs. SP
TCP vs. DBCP	GHB vs. SP

Table 5. Previous comparisons.

Did the authors compare their ideas? Table 5 shows which mechanism has been compared against which previous mechanisms (listed in chronological order). Few arti-

[4] Note however, that we did not evaluate power consumption due to off-chip accesses and which usually accounts for a majority of power expenses.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

cles have quantitative comparisons with (one or two) previous mechanisms, except when comparisons are almost compulsory, like *GHB* which compares against *SP* because it is based on this mechanism. Sometimes, comparisons are performed against the most recent mechanisms, like *TCP* and *TK* which are compared against *DBCP*, while in this case, a comparison with *SP* might have been more appropriate.

	Base	TP	VC	SP	Markov	FVC	DBCP	TKVC	TK	CDP	CDPSP	TCP	GHB
1		√	√	√	√	√	√	√	√	√	√		√
2		√	√	√	√	√	√	√	√	√	√		√
3		√	√	√	√	√	√	√	√		√		√
4		√	√	√	√	√	√	√	√		√		√
5		√	√	√	√	√	√	√	√		√		√
6		√	√	√	√	√		√	√		√		√
7		√	√	√	√	√		√	√		√		√
8		√	√	√	√	√		√	√		√		√
9		√	√	√	√	√		√	√		√		√
10		√	√	√		√		√	√		√		√
11		√	√	√		√		√	√		√		√
12		√	√	√		√		√	√		√		√
13		√	√	√				√	√		√		√
14		√	√	√				√	√		√		√
15		√	√	√				√	√		√		√
16		√	√	√				√	√		√		√
17		√		√				√	√		√		√
18		√		√				√	√		√		√
19		√		√				√	√		√		√
20				√					√		√		√
21				√					√		√		√
22				√					√		√		√
23				√									√
24													√
25													√
26													√

Table 6. Which mechanism can be the best with N benchmarks?

3.2 What is the effect of benchmark selection on ranking?

Yes, cherry-picking can be wrong. We have ranked the different mechanisms for every possible benchmark combination, from 1 to 26 benchmarks (there are 26 Spec benchmarks). First, we have observed that for any number of benchmarks less or equal than 23, i.e., the average IPC is computed over 23 benchmarks or less, there is always *more than one winner*, i.e., it is always possible to find two benchmark selections with different winners. In Table 6, we have indicated how often a mechanism can be a winner for any number N of benchmarks from $N = 1$ to $N = 26$ (i.e., is there an N-benchmark selection where the mechanism is the winner?). For instance, mechanisms that perform poorly on average, like *FVC*, can win for selections of up to 12 benchmarks; note that *CDP* is a prefetcher for pointer-based data structures, so that it is likely to perform well for benchmarks with many misses in such data structures; for the same reason, *CDPSP* (a combination of *SP* and *CDP*) can be appropriate for a larger range of benchmarks, as the authors point out. Another astonishing result is *Markov* which

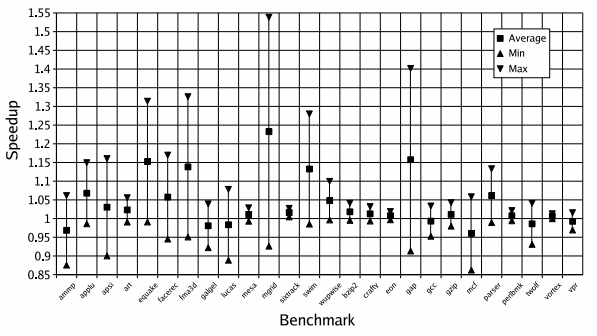

Figure 6. Benchmark sensitivity.

can perform well for up to 9-benchmark selections, because its effect varies strongly from one benchmark to another, depending on the regularity of access patterns (hence the poor average performance); for instance, *Markov* outperforms all other mechanisms on *gzip* and *ammp*.

Note that not using the full benchmark suite can be naturally profitable, but also detrimental, to the proposed mechanism. In Table 7, we have indicated the ranking with all 26 Spec benchmarks, and with the benchmark selections used in the *DBCP* and *GHB* articles. While *DBCP* favors its article benchmark selection, *GHB* performs better when considering all 26 benchmarks rather than its article benchmark selection, and for which it is outperformed by *SP*.

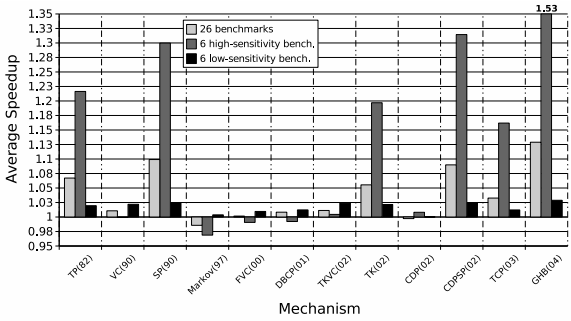

Figure 7. High- and low-sensitivity benchmarks speedup.

More generally, the benchmark sensitivity to mechanisms varies greatly, as shown in Figure 6. Obviously, some benchmarks are barely sensitive to data cache optimizations like *wupwise, bzip2, crafty, eon, perlbmk* and *vortex*, while others, like *apsi, equake, fma3d, mgrid, swim* and *gap* will have a strong impact on assessing research ideas. To further demonstrate this benchmark sensitivity, Figure 7 shows the performance and classification of all mechanisms using the 26 benchmarks, the 6 high-sensitivity benchmarks and the 6 low-sensitivity benchmarks. Obviously, absolute observed performance and ranking are severely affected by the benchmark selection.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

	Base	TP	VC	SP	Markov	FVC	DBCP	TKVC	TK	CDP	CDPSP	TCP	GHB
26 benchmarks	11	6	8	2	13	10	9	7	4	12	3	5	1
DBCP benchmark selection	8	13	4	1	9	7	3	6	5	12	10	11	2
GHB benchmark selection	10	13	8	1	11	9	6	7	4	12	3	5	2

Table 7. Influence of benchmark selection.

3.3 What is the effect of the architecture model precision on ranking?

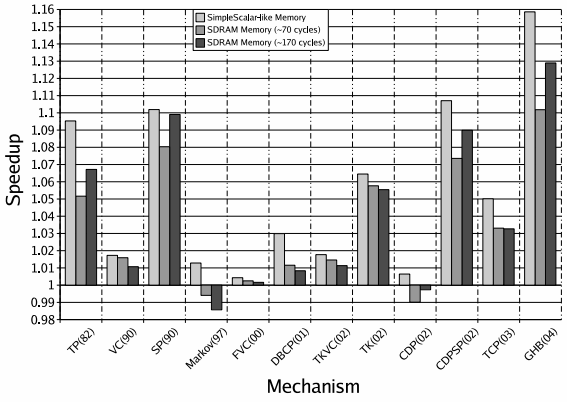

Figure 8. Effect of the memory model.

Is it necessary to have a more detailed memory model? We have implemented a detailed SDRAM model, as Cuppu et al. [5] did for SimpleScalar, and we have evaluated the influence of the memory model on ranking. The original SimpleScalar memory model is rather raw with a constant memory latency. Our model uses a bank interleaving scheme [20, 30] which allows the DRAM controller to hide the access latency by pipelining page opening and closing operations. We implemented several schedule schemes proposed by Green et al. [8] and retained one that significantly reduces conflicts in row buffers.

Figure 8 compares the 170-cycle SDRAM used in our experiments (see Table 1), with a constant 70-cycle SimpleScalar-like memory used in many articles, and also with an SDRAM exhibiting an average 70-cycle latency (like the SimpleScalar memory); to achieve that latter SDRAM latency, we have scaled down the original SDRAM parameters (especially the CAS latency, which was reduced from 6 to 2 memory cycles). The memory model can significantly affect the absolute performance as well as the ranking of the different mechanisms. Overall, speedups are reduced by 57.9% in average from the SimpleScalar-like memory model to the 170-cycle SDRAM memory model, and 59.9% to the 70-cycle SDRAM memory model. The performance of *GHB* and *SP* are respectively reduced by 18.7% and 2.8%, and the greater reduction of *GHB* performance is due to the fact that *GHB* increases memory pressure and is therefore sensitive to stricter memory access rules. And the more precise memory model also affects ranking: for instance, *DBCP* clearly outperforms *VC* and *TKVC* with a SimpleScalar-like memory model, while *VC* and *TKVC* outperform *DBCP* with an SDRAM memory.

Unlike in the SimpleScalar memory model, the SDRAM average memory latency varies strongly from one benchmark to another, e.g., from 87.42 (*gzip*) processor cycles to 389.73 (*lucas*) processor cycles for the baseline cache configuration. The average latency discrepancy is due to page modes and burst SDRAM requests which both favor long sequences of consecutive accesses to the same row. For the same reason, there is a large average memory latency discrepancy among mechanisms because they induce different memory request patterns. For instance, on *lucas*, the average SDRAM memory latency is 389 processor cycles for the baseline cache configuration, and 490 processor cycles for *GHB* due to the increased number of accesses (13 million for the baseline configuration and 17 million for *GHB*) and the increased number of precharges (from 10 million to 13 million) for a program that was already memory-bound. As a result, the bus stalls more often, inducing a slowdown of 0.76 on this benchmark, against a 1.12 speedup with the SimpleScalar memory model. Naturally, such discrepancies particularly show for mechanisms which strongly affect memory requests, especially *TP* and *CDPSP* (prefetching), and less so for *VC* and *TKVC* (victim cache).

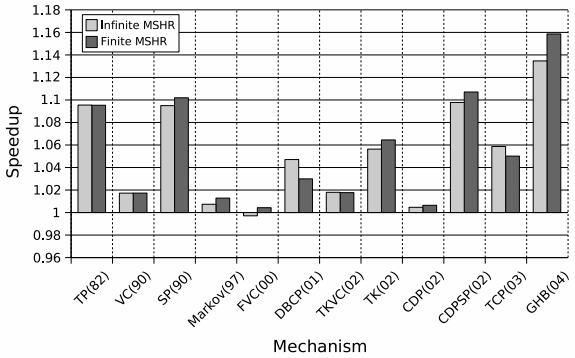

Figure 9. Effect of the cache model accuracy.

Influence of cache model inaccuracies. Similarly, we have investigated the influence of other hierarchy model components. For instance, we have explained in Section 2.2 that the SimpleScalar cache uses an infinite miss address file (MSHR), so we have compared the effect of just varying

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

the miss address file size (i.e., infinite versus the baseline value defined in Table 1).

Figure 9 shows that for many mechanisms, the MSHR has a limited but sometimes peculiar effect on performance, and it can affect ranking. Surprisingly, several mechanisms seem to perform better with a finite-sized MSHR than with an infinite one. For instance, *TCP* outperforms *TK* with an infinite MSHR, but not with a finite MSHR, because then, when the MSHR is full, the cache is stall, the bus is not used, giving more opportunities to *TK* (located in the L1 cache, versus the L2 cache for *TCP*) to send prefetch requests.

3.4 What is the effect of second-guessing the authors' choices?

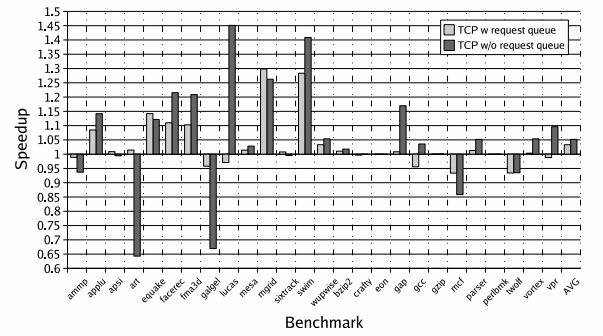

Figure 10. Effect of second-guessing.

For several of the mechanisms, some of the implementation details were missing in the article, or the interaction between the mechanisms and other components were not sufficiently described, so we had to second-guess them. While we cannot list all such omissions, we want to illustrate their potential effect on performance and ranking, and that they can significantly complicate the task of reverse-engineering a mechanism.

One such case is *TCP*; the article properly describes the mechanism, how addresses are predicted, but it gives few details on how and when prefetch requests are sent to memory; actually, many articles dealing with prefetch mechanisms similarly omitted to describe this part of their implementation. Among the many different possibilities, prefetch requests can be buffered in a queue until the bus is idle and a request can be sent. Assuming this buffer effectively exists, a new parameter is the buffer size; it can be either 1 or a large number (we ended up using a 128-entry buffer), and the buffer size is a tradeoff, since a too short buffer size will result in the loss of many prefetch requests (they have to be discarded), and a too large one may excessively delay some prefetch requests. Figure 10 shows the performance difference and ranking for a 128-entry and a 1-entry buffer. All possible cases are found: for some benchmarks like *crafty* and *eon*, the performance difference is tiny, while it is dramatic for *lucas*, *mgrid* and *art*. For instance, the performance of *lucas* decreases (with a 128-buffer) because a large prefetch buffer always contains pending prefetch requests and will seize the bus whenever it is available, increasing the probability that normal miss requests are delayed.

We ended up selecting a 128-buffer because it matched best the average performance presented in the article. Later on, when we contacted the *TCP* authors, we found that they did have such a buffer; though we could not get a confirmation of the prefetch buffer size at the time of publication.

3.5 What is the effect of trace selection on ranking?

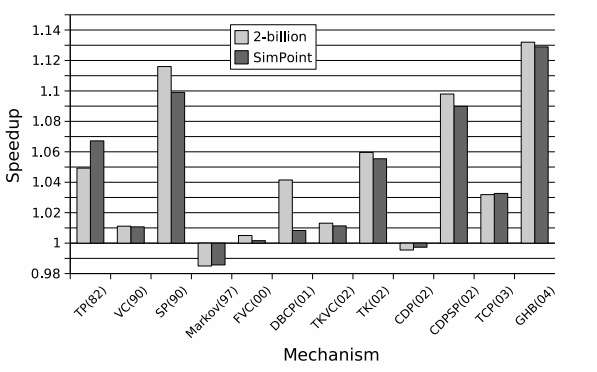

Figure 11. Effect of trace selection.

Most researchers tend to skip an arbitrary (usually large) number of instructions in a trace, then simulate the largest possible program chunk (usually of the order of a few hundred million to a few billion instructions), as for *TK*, *TCP* and *DBCP* for instance: skip 1 billion instructions and simulate 2 billion instructions. Sampling has received increased attention in the past few years [26, 21], with the prospect of finding a robust and practical method for speeding up simulation while ensuring the representativity of the sampled trace. One of the most notable and practical contributions is SimPoint [21] which shows that a small trace can highly accurately emulate a whole program behavior.

In order to evaluate the effect of trace selection, we compared the arbitrary "skip 1 billion, simulate 2 billion" traces with 500-million SimPoint traces (the ones used throughout the article). Figure 11 shows the average performance achieved with each method, and they differ significantly. Overall, most mechanisms appear to perform better with an arbitrary 2-billion trace, with the notable exception of *TP*. Not surprisingly, trace selection can have a considerable effect on *research* decisions like selecting the most appropriate mechanism, and obviously, even 2-billion traces (which are considered reasonably large) apparently do not constitute a sufficient precaution.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

4 *MicroLib*

MicroLib. A major goal of MicroLib is to build an open library of processor simulator components which researchers can easily download either for plugging them directly in their own simulators, or at least for having full access to the source code, and thus to a detailed description of the implementation. There already exists libraries of open simulator components, such as OpenCores [2], but these simulators are rather IP blocks for SoC (System-on-Chip), i.e., an IP block is usually a small processor or a dedicated circuit, while MicroLib aims at becoming a library of (complex) processor subcomponents, and especially of various *research* propositions for these processor components.

Our goal is to ultimately provide researchers with a sufficiently large and appealing collection of simulator models that researchers actually start using them for performance comparisons, and more importantly, that they later on start contributing their own models to the library. As long as we have enough manpower, we want to maintain an up-to-date comparison (ranking) of hardware mechanisms, for various processor components, on the MicroLib web site. That would enable authors to demonstrate improvements to their mechanisms, to fix mistakes a posteriori, and to provide the community with a clearer and fair comparison of hardware solutions for at least some specific processor components or research issues.

MicroLib and existing simulation environments. MicroLib modules can be either plugged into MicroLib processor models (a superscalar model called OoOSysC and a 15% accurate PowerPC750 model are already available [16]) which were developed in the initial stages of the project, or they can be plugged into existing processor simulators. Indeed, to facilitate the widespread use of MicroLib, we intend to develop a set of *wrappers* for interconnecting our modules with existing processor simulator models such as SimpleScalar, and recent environments such as Liberty [25]. We have already developed a SimpleScalar wrapper and all the experiments presented in this article actually correspond to MicroLib data cache hardware simulators plugged into SimpleScalar through a wrapper, rather than to our own superscalar model. Next, we want to investigate a Liberty wrapper because some of the goals of Liberty fit well with the goals of MicroLib, especially the modularity of simulators and the planned development of a library of simulator modules. Rather than competing with modular simulation environment frameworks like Liberty (which aim at providing a full environment, and not only a library), we want MicroLib to be viewed as an open and, possibly federating, project that will try to build the largest possible library through extensive wrapper development. There are also several modular environments in the industry, such as ASIM [6] by Compaq (and now Intel), TSS by Philips, and though they are not publicly available, they may benefit from the library, provided a wrapper can be developed for them. The current MicroLib modules are based on *SystemC* [18], a modular simulation framework supported by more than 50 companies from the embedded domain, which is quickly becoming a *de facto* standard in the embedded world for cycle-level or more abstract simulation. All the mechanisms presented in this article were implemented using *SystemC*.

5 Conclusions and Future Work

In this article we have illustrated, using data caches, the usefulness of a fair quantitative comparison of hardware optimizations. We have implemented several recent hardware data cache optimizations and we have shown that many of current and widespread methodology practices (benchmark selection, trace selection, inaccurate memory models, reverse-engineering research articles,...) can result in the incorrect assessment of what is the best or most appropriate mechanism for a given architecture. Such results suggest improved evaluation methodology is badly needed, especially the disclosure of simulators at the time of publication, and interoperable simulators. In that spirit, we are developing a library of modular simulator components, called MicroLib, aiming at promoting the sharing, and reuse of simulators. Our goal is now to populate the library, to encourage the quantitative comparison of mechanisms, and to maintain a regularly updated comparison (ranking) for various hardware components.

6 Acknowledgements

We would like to thank all the authors with whom we have interacted; we are especially grateful to Babak Falsafi and Mike Ferdman for their help on *DBCP*. We also would like to thank the reviewers, and the members of the INRIA Alchemy group for their many helpful comments and suggestions.

References

[1] MicroLib. http://www.microlib.org.
[2] OPENCORES. http://www.opencores.org, 2001-2004.
[3] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Sciences, University of Wisconsin, June 1997.
[4] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 279–290, San Jose, California, October 2002.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**

[5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th annual international symposium on Computer architecture (ISCA)*, pages 222–233, Atlanta, Georgia, United States, June 1999.

[6] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukkerjee, H. Patil, S. Wallace, N. Binkert, and T. Juan. ASIM: A performance model framework. In *IEEE Computer, Vol. 35, No. 2*, February 2002.

[7] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110. IEEE Computer Society Press, 1992.

[8] C. Green. Analyzing and implementing SDRAM and SGRAM controllers. In *EDN (www.edn.com)*, February 1998.

[9] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA)*, pages 209–220, Anchorage, Alaska, May 2002.

[10] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag correlating prefetchers. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, Anaheim, California, February 2003.

[11] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *International Symposium on Low Power Electronics and Design (ISLPED 01)*, Huntington Beach, California, August 2001.

[12] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, pages 252–263, Denver, Colorado, United States, June 1997.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. Technical report, Digital, Western Research Laboratory, Palo Alto, March 1990.

[14] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, pages 144–154, Gteborg, Sweden, June 2001.

[15] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 11–21. ACM Press, 2000.

[16] G. Mouchard. PowerPC G3 simulator. http://www.microlib.org/G3/PowerPC750.php, 2002.

[17] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, page 96, Madrid, Spain, February 2004.

[18] OSCI. SystemC. http://www.systemc.org, 2000-2004.

[19] R. Rakvic, B. Black, D. Limaye, and J. P. Shen. Non-vital loads. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*. ACM Press, 2002.

[20] T. Rockicki. Indexing memory banks to maximize page mode hit percentage and minimize memory latency. Technical report, HP Laboratories Palo Alto, June 1996.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, pages 45–57. ACM Press, 2002.

[22] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical report, HP Laboratories Palo Alto, August 2001.

[23] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[24] SPEC. SPEC2000. http://www.spec.org.

[25] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, November 2002.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

[27] J. Yang and R. Gupta. Energy efficient frequent value data cache design. In *Proceedings of the 35th international symposium on Microarchitecture (MICRO)*, pages 197–207, Istanbul, Turkey, November 2002.

[28] Y. Zhang and R. Gupta. Enabling partial cache line prefetching through data compression. In *International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003.

[29] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, pages 150–159, Cambridge, Massachusetts, United States, November 2000.

[30] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd international symposium on Microarchitecture (MICRO)*, Monterey, California, December 2000.

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 **IEEE**